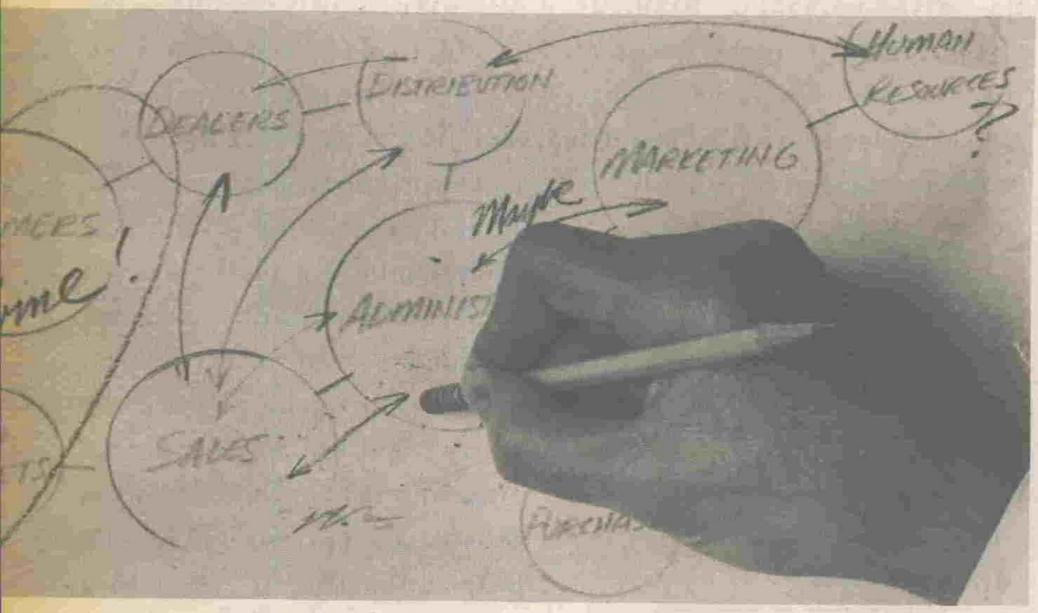
กว่าจะเขียนโปรแกรมให้ถูกวิธี สอนกันได้ไม่ยาก



ปัญหาโลกแตกที่โปรแกรมเมอร์ทั้งหลายมักประสบกันเป็นประจำในงานดาต้าโปรเซสซิ่ง แต่ใช่ว่าจะไม่มีทางออกซะเลย เคล็ดลับที่บางคนมองข้าม

ปัญหาเรื่องการพัฒนาโปรแกรมได้ช้า ทำแล้วมีบั๊กต้องแก้ไขมาก นับเป็นปัญหาใหญ่ที่สุดในงานดาต้าโปรเซสซิ่ง มีคนพยายามแก้ปัญหานี้โดยคิดค้นวิธีการต่าง ๆ เช่น การทำ Structured Programming และ Modular Design บางคนคิดเลยไปจนถึงขั้นปฏิรูปการเขียนโปรแกรมโดยสร้างภาษาใหม่แบบ Non-procedural บางทีก็เรียกว่า Fourth Generation Language แต่ลงท้ายทุกคนเริ่มหันกลับมาดูกันจริง ๆ แล้วสรุปว่า การเขียนโปรแกรมให้ดีนั้นต้องเริ่มจากการพิจารณาโครงสร้างข้อมูลที่จะใช้ในงานนั้น จัดลำดับขั้นตอนการกระทำต่อข้อมูลนั้น ๆ เราเรียกขั้นตอนเหล่านี้ว่า อัลกอริทึม (Algorithm) ถ้าภาพของขั้นตอนเหล่านั้นกระจ่างแจ้งอยู่ในสมองของเราแล้ว การแปลเป็นภาษาโปรแกรมก็ง่าย สามารถแก้ไขดัดแปลงในภายหลังได้ไม่ยาก การตรวจสอบความถูกต้องตลอดจนการดีบั๊กก็ทำได้รวดเร็ว และเมื่อผ่านการทดสอบเรียบร้อยเราก็จะได้โปรแกรมที่น่าเชื่อถือเพื่อการใช้งานต่อไป

แค่ฟังดูมันก็ง่ายดี แต่ติดปัญหาว่าการสอนคนให้คิดและวางขั้นตอน (Algorithm) ให้ถูกต้องนั้นสอนกันได้ยากมาก ในมหาวิทยาลัยต่าง ๆ มีวิชาในลักษณะดังกล่าวอยู่พอสมควรแต่ขาดผู้สอน ผมเคยสอนวิชา Computer Algorithm อยู่หลายปี มีลูกศิษย์จบออกไปหลายรุ่น ผมสอบถามพวกเขาเสมอ ๆ ว่า *"ก่อนเข้าเรียนวิชานี้ทราบหรือไม่ว่าเราจะพูดถึงเรื่องอะไร"* และหลังจากเรียนจบแล้วได้งานทำจนเป็นใหญ่เป็นโต ผมเคยลองถามอีกว่า *"คุณได้ใช้ประโยชน์จากวิชานี้แค่ไหน"*

คำตอบก็คือ *"ก่อนเข้าเรียนไม่รู้ว่าจะเป็นเรื่องอะไร เรียนจบแล้วไม่แน่ใจว่าจะได้ประโยชน์อะไร พอทำงานสัก 3-5 ปีจึงค่อย ๆ จะเริ่มเข้าใจ และสรุปลงท้ายว่าพวกเราโชคดีมากที่ได้เรียนวิชานี้"*

ผมทราบดีว่าเรื่องอย่างนี้เป็นการปิดทองใต้ฐานพระ (หลังพระยังมีคนเห็น) ในระยะ 10 ปีให้หลังมานี้ผมไม่ได้สอนอีกเลย เพราะมองไม่เห็นผลตอบแทน แต่อย่างไรก็ตามการทำงานด้านดาต้าโปรเซสซิ่งนี้เราย่อมหนีไม่พ้น และต้องสร้างคนเพื่อช่วยกันทำงาน ความจำเป็นมันบีบบังคับให้เราย้อนกลับมาทำหน้าที่นี้อยู่เสมอ ๆ ผมอยากจะเล่าให้ฟังว่าปัญหาการสอนคน (เขียนโปรแกรม) ให้ถูกวิธี) นี้เป็นอย่างไร

รู้จักกับอัลกอริทึมกันก่อน

อัลกอริทึมคือคำบรรยายเพื่อบอกให้คนทำตามเป็นลำดับขั้น ซึ่งเมื่อทำเสร็จจนจบแล้วจะได้ผลงานอย่างใดอย่างหนึ่งดังที่เราตั้งใจไว้ ยกตัวอย่างเช่น จะหาตัวหารร่วมมาก (ห.ร.ม.) ของตัวเลข 2 ตัวมีหลักเกณฑ์ให้ทำเป็นขั้นตอนดังนี้

ก. เอาตัวที่มากกว่าเป็นตัวตั้งและตัวที่น้อยกว่าเป็นตัวหาร
ข. ทำการหารกันแล้วหาเศษของการหารนั้น
ค. ถ้าเศษเป็นเลขศูนย์ ตัวที่เล็กกว่านั้นคือ ห.ร.ม.
ง. ถ้าหารแล้วเศษไม่เป็นศูนย์ ให้เอาเศษนั้นไปแทนที่ตัวที่ใหญ่กว่า
จ. กลับไปทำตามคำสั่งตั้งแต่ข้อ ก. ใหม่อีก

อัลกอริทึมข้างต้นนี้ออกแบบโดยยูคลิด ซึ่งเป็นผู้ให้กำเนิดวิชาเรขาคณิต อันที่จริงคงจะมีคนพูดกันมาก่อนแล้ว ส่วนยูคลิดนำมาตีแผ่ให้เห็นถึงแก่น ข้อดีของมันก็คือ ปกติคนธรรมดา ๆ สามารถทำการบวก ลบ คูณ และหารได้ (อย่างถูกต้องนะครับ) แต่เราทำให้คน ๆ นั้นมีความสามารถที่เหนือชั้นขึ้นมาได้อีกขั้นหนึ่งคือ คำนวณค่า ห.ร.ม.ได้ จากจุดนี้เราก็ทำการคิดค้นสร้างอัลกอริทึมสำหรับหารูตที่ 2 รูตที่ 3 ตัวควบ ฯลฯ ได้เช่นกัน และถ้าจะคิดย้อนกลับไปอีกนิดหนึ่งถึงเด็กที่เพิ่งเริ่มเรียนรู้วิธีการบวกและลบตัวเลข เราก็สามารถจะสร้างอัลกอริทึมสำหรับการคูณ การหารยาว การหารสั้น ฯลฯ ซึ่งภายหลังต่อมาก็กลายเป็นความรู้ขั้นพื้นฐานสำหรับสอนเด็ก พูดง่าย ๆ คือตั้งแต่เราเกิดมา ผู้ใหญ่ก็ได้สอน *"อัลกอริทึม"* ต่าง ๆ ให้เรามาตลอด จนกระทั่งเราสามารถทำการคำนวณอย่างสลับซับซ้อนได้ (แต่มาตายตรงที่ว่าจะสอนการเขียนโปรแกรมให้ดีกลับทำไม่ได้ !)

พึงสังเกตว่าอัลกอริทึมนี้บรรยายเป็นภาษาคน และเดิมก็มุ่งเน้นที่จะสอนให้คนทำตาม แต่แนวคิดเรื่องนี้ได้ขยายไปสู่งานคอมพิวเตอร์ เพราะเครื่องคอมพิวเตอร์คือเครื่องมือชนิดหนึ่งที่สามารถทำงานตามคำสั่งขั้นพื้นฐานจำนวนหนึ่งได้อย่างถูกต้องและแม่นยำ ดังนั้นเราจึงสร้างอัลกอริทึมที่เป็นภาษาคอมพิวเตอร์ให้เครื่องคอมพิวเตอร์ปฏิบัติตามได้เช่นกัน ผลที่ได้ก็คือโปรแกรม โปรแกรมจึงเป็นอัลกอริทึมชนิดหนึ่ง

สร้างอัลกอริทึมให้ถูกวิธี

การสร้างอัลกอริทึมนั้นเราต้องวางเป้าหมาย เช่น จะหาค่า ห.ร.ม. หรือค่า ค.ร.น. หรือรูตที่ 2 ฯลฯ อัลกอริทึมต้องมีจุดเริ่มต้น และเมื่อทำไปแล้วต้องไปให้ถึงจุดจบอันหนึ่ง ณ จุดจบดังกล่าวเราจะต้องได้ผลงานตามที่ตั้งใจไว้ ตัวอย่างในอัลกอริทึมของยูคลิดนั้น ข้อ ก. คือจุดเริ่มต้น ข้อ ค. คือจุดจบ เมื่อใดที่หารกันแล้วเศษเป็นเลขศูนย์เราก็จะได้คำตอบ

การเรียบเรียงคำสั่งให้เป็นอัลกอริทึมนี้เรามักพบข้อผิดพลาดได้ หมายถึง

ว่าคอมพิวเตอร์ทำตามคำสั่งที่เราเรียบเรียงขึ้นมาไม่ได้ผล อาจไปสิ้นสุดที่ไหนสักแห่งหนึ่งตามสั่ง แต่ได้คำตอบมาผิด หรืออาจวิ่งวนเวียนไม่รู้จบก็ได้ ปัญหานี้จะเกิดขึ้นในกรณีที่เป้าหมายของเรายากเกินไป เป้าหมายที่ต้องใช้คำสั่งหรือขั้นตอนนับสิบนับร้อยคำสั่งย่อมมีโอกาสผิดพลาดได้ ปัญหานี้แก้ได้โดยวิธีสร้างเป้าหมายย่อย แต่ละเป้าหมายย่อยนั้นถ้าสามารถทำได้ด้วยอัลกอริทึมที่ยาวประมาณ 10-50 บรรทัด เราจะเขียนให้ถูกต้องได้ง่าย ยกตัวอย่างเช่น การสร้างโปรแกรมให้คอมพิวเตอร์เดินหมากรุกนั้นอาจต้องแยกเป็น 4 ขั้นตอนใหญ่ดังนี้

ก. คำนวณว่ามีหนทางเดินทั้งหมดกี่หนทาง
ข. คำนวณค่าการได้กินและการเสียตัวหมากที่จะเกิดขึ้นของแต่ละหนทางเดินในข้อ ก.
ค. เลือกหนทางที่ดีที่สุดตามค่าในข้อ ข.
ง. ส่งผลให้แก่รูทีนในการเดินตัวหมากรุก

จากเป้าหมายใหญ่เหล่านี้ถ้าเราดูแล้วเห็นว่า แต่ละเป้าหมายใหญ่ยังยากเกินกว่าที่จะคิดวางเป็นคำสั่งด้วยภาษาคอมพิวเตอร์ เราอาจแยกย่อยลงไปอีกได้ เช่น ข้อ ก. ควรเช็คเสียก่อนว่าเป็นตัวอะไร (ม้า, เรือ, ขุน ฯลฯ) แล้วจึงแยกไปพิจารณาตาเดินเป็นแบบ ๆ ไป ด้วยวิธีนี้เราสามารถทำให้แต่ละเป้าหมายย่อยสามารถสร้างคำสั่งได้โดยไม่ผิดพลาด

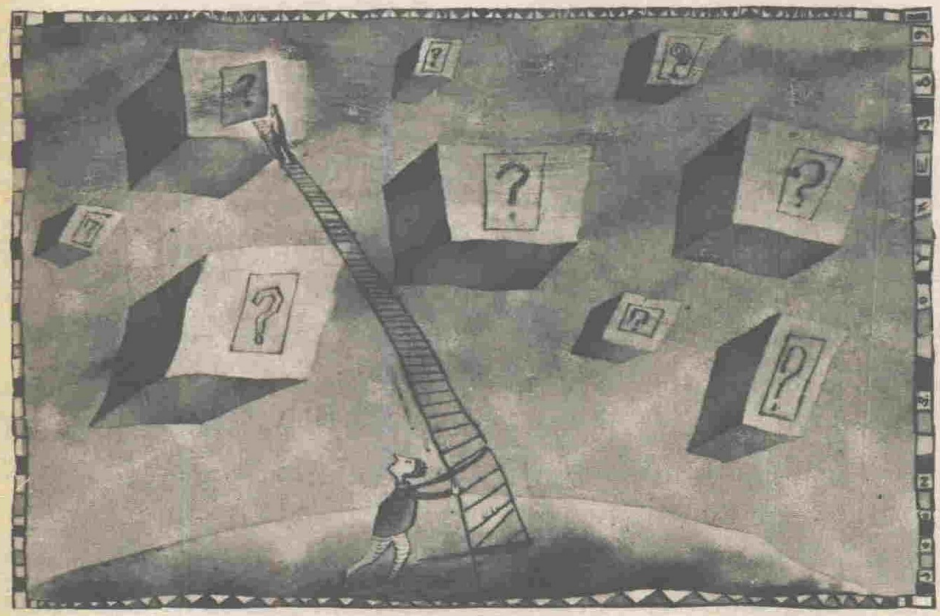
เลือกไม่เป็น (Bad Choice)

การแยกเป้าหมายใหญ่ออกเป็นเป้าหมายย่อย และการเลือกคำสั่งมาเรียบเรียงเพื่อให้ได้ผลตามเป้าหมายย่อยต่าง ๆ นั้น เราทำได้มากมายหลายวิธี และนี่คือจุดที่จะแยกคนฝีมือดีออกจากคนฝีมือธรรมดา ๆ คนฝีมือดีจะรู้ว่าควรแยกเป็นเป้าหมายอย่างไรจึงจะเลือกคำสั่งได้ง่ายและมีโอกาสพลาดน้อย

พึงสังเกตว่าในขั้นตอนการแยกออกเป็นเป้าหมายย่อยนี้ บางครั้งเรายังไม่สามารถบอกได้ว่าคำสั่งที่ประกอบกันขึ้นมาเพื่อทำให้สำเร็จตามเป้าหมายย่อยแต่ละส่วนจะเป็นอย่างไร ต่อให้เก่งกาจขนาดไหนก็ไม่สามารถจะบอกได้ทันที แต่มันอาจเกิดจากความรู้สึกหรือความเชื่อบางประการว่าทำแบบนั้นแหละดี เราเรียกความสามารถชนิดนี้ว่า *"สัมผัสที่ 6"* ก็น่าจะได้ คนฝีมือดีจะมีสัมผัสที่ 6 เป็นตัวชี้แนะภายในจิตใต้สำนึก ช่วยให้พวกเขากระทำผิดพลาดน้อย ถึงแม้จะค้นพบในภายหลังว่าเขาคิดขาดหายไปบางเรื่อง แต่เขาก็เพิ่มเติมได้โดยไม่เปลี่ยนเค้าโครงหรือขั้นตอนหลักเหล่านั้น

ในทางตรงกันข้าม สำหรับคนที่มีฝีมือปานกลางมักจะแยกเป้าหมายหลักออกเป็นเป้าหมายย่อยที่สับสน และเป็นเป้าหมายย่อยที่ทำให้สำเร็จได้ยาก ในระหว่างการเรียบเรียงคำสั่งเป็นอัลกอริทึมสำหรับเป้าหมายย่อยอันใดอันหนึ่งเขาอาจจะประสบปัญหาจนถึงกับต้องเปลี่ยนเป้าหมาย ซึ่งส่งผลกระทบต่อหลักการใหญ่ที่เป็นตัวจัดแยกเป้าหมายย่อย การแก้ไขหลักการในขั้นตอนหนึ่งทำให้เขาต้องไปดัดแปลงอัลกอริทึมส่วนอื่นที่วางไว้ หรือลืมแก้ไขให้สอดคล้องกันก่อนทดสอบโปรแกรม โปรแกรมนั้นจึงทำงานไม่ได้ผลสักที

จากความจริงที่กล่าวมาทั้งหมดนี้ เราจะเห็นว่าการมอบหมายให้คนที่ขาดประสบการณ์คิดหาทางแยกเป้าหมายเอง เรียบเรียงคำสั่งเองนั้นมันเป็นไปได้ยาก คนจะต้องมี *"สัมผัสที่ 6"* เสียก่อน ซึ่งเราก็ยังไม่รู้แน่ชัดว่าสัมผัสที่ 6 ดังกล่าวนี้เป็นอย่างไร มันเป็นเสมือน *"พรสวรรค์"* สำหรับคนบางคนเท่านั้น และมิใช่ว่าจะสร้างกันได้ง่าย ๆ เท่าที่ผ่านมาผมสามารถสอนให้คนจำนวนหนึ่งเกิด *"ปิ๊ง"* ขึ้นมาได้ในทันที แต่อีกจำนวนหนึ่งต้องใช้วิธีหยิบยกหลาย ๆ ตัวอย่างมาขัดเยียดลงไปในหัวของเขา วิธีนี้ช่วยให้เขา



ทำงานแบบพอไปได้ ต้องเป็นงานประจำที่เราเคยสอนไว้เท่านั้น หากเมื่อใดเจองานรูปแบบใหม่เขาอาจงมอยู่เป็นเดือน ๆ อีก

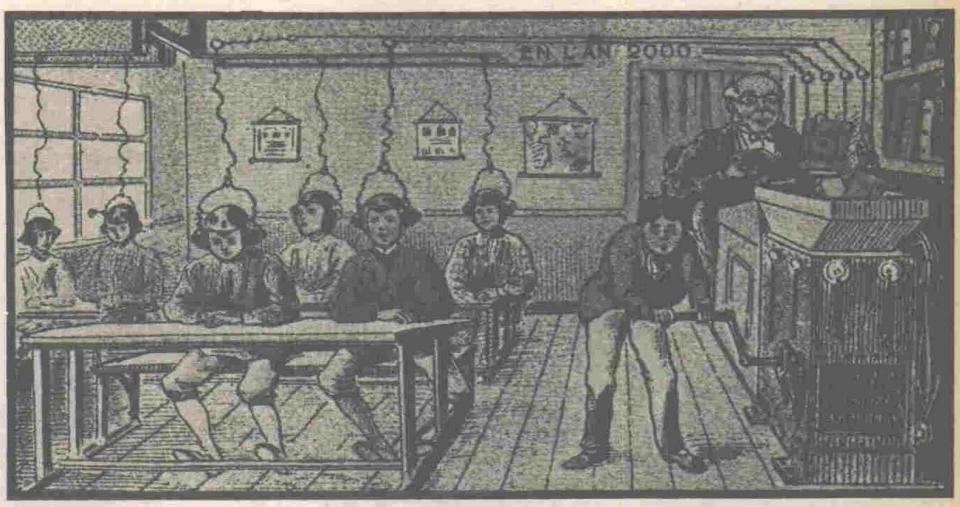
การยึดติดมั่น (Fixed Idea)

การทำความเข้าใจกับแนวคิดของคนอื่นนับเป็นเรื่องยากสำหรับมนุษย์เรา ยิ่งเป็นเรื่องที่คน ๆ นั้นระดมความคิดด้วยตัวเองมาเป็นเวลาหนึ่งแล้วล่ะก็เขาจะยิ่งเข้าใจ *"วิธีอื่น"* ได้ยากขึ้น เราเรียกสภาพการณ์อย่างนี้ว่า เกิดการยึดติดมั่น (Fixed Idea) คำภาษาอังกฤษให้ความหมายได้ดีมาก คือเมื่อคนเราใช้เวลาคิดหาวิธีด้วยตนเองไปสักพักหนึ่งแล้วเขาจะเกิดความยึดมั่น (Fixed) ติดกับแนวคิด (Idea) ที่เขาเห็นว่าดีที่สุด แต่ปัญหามีอยู่ว่า วิธีการหรือแนวความคิดที่เขาสร้างขึ้นมานั้นมันอาจไม่ใช่วิธีที่ดี หรือกลายเป็นวิธีที่ไม่ถูกต้อง ไม่สมบูรณ์ ทำแล้วจะติดปัญหา โปรแกรมไม่ทำงานตามที่ตั้งใจไว้ หากปล่อยจนถึงจุดนี้ปัญหาคงจะยุ่งยากมากถ้าเราเป็นที่ปรึกษา เพราะไม่ว่าเราจะเสนอแนะแนวความคิดใหม่ หรือชี้แจงว่าแนวความคิดเก่าของเขาผิดพลาดตรงไหน เขาจะไม่ยอมเชื่อฟังเลย เท่ากับเป็นการปิดประตูไม่ให้ใครช่วยเขาได้เลย

ปัญหาการยึดติดมั่นมีอยู่ในมนุษย์เกือบทุกรูปทุกนาม แต่ในเรื่องของการเขียนโปรแกรมนี้อาจพบบ่อยมาก คนส่วนใหญ่เมื่อเริ่มเขียนโปรแกรมไปแล้วจะไม่อยากคิดทางอื่น เขาจะพยายามแก้ไขต่อไปเรื่อย ๆ จนกว่าจะเสร็จ ยิ่งทำมากก็ยิ่งยึดมั่น จึงกลายเป็นความรู้สึกประเภท *"เราถอยไม่ได้แล้ว"*

หนทางแก้การยึดติดมั่น

สิ่งที่ผู้บริหารหรือหัวหน้าทีมพึงระมัดระวังก็คือ อย่าให้เกิดการยึดติดมั่น งานไหนที่มีลักษณะใหม่ต้องบุกเบิกหรือเกินความสามารถของลูกทีม เราควรมอบหมายให้พนักงานฝีมือดีวางโครงร่างคร่าว ๆ เสียก่อน เปิดโอกาสซักซ้อมทำความเข้าใจกันจนเป็นที่เรียบร้อยแล้วจึงค่อยเริ่มเขียนโปรแกรม



นอกจากจะป้องกันมิให้คนของเราเกิดการยึดมั่นกับวิธีที่ผิดแล้ว เราต้องฝึกฝนให้รู้จักแยกงานใหญ่เป็นงานย่อย ในการฝึกฝนดังกล่าวนี้จุดที่เน้นหนักก็คือ ดูว่าแต่ละวิธีการแยกงานนั้นเราจะนำตัวเราไปสู่ปัญหาอื่น ๆ ได้มากน้อยแค่ไหน การประเมินสถานการณ์หรือการคาดการณ์ล่วงหน้านับเป็นเรื่องที่สำคัญที่สุด คนส่วนใหญ่จะพยายามเลือกทิศทางที่จะทำให้โปรแกรมสั้น หรือไม่ก็ให้ดูหรูหราพิสดาร เขาจะให้ค่ากับปัญหาที่อาจเกิดขึ้นในทางต่ำ และยอมแลกในทิศทางที่สวนกับสิ่งที่ควรจะเป็น นั่นคือ ถึงแม้เขารู้ว่าจะเกิดปัญหามากมาย แต่ถ้าโปรแกรมดูวิลิศมาหราขึ้นเขาก็ยินยอม เรื่องแบบนี้ต้องสร้างความเข้าใจส่วนบุคคล ใช้ประสบการณ์ของพวกเขาเป็นตัวชี้ว่า ปัญหาที่เกิดขึ้นนั้นสร้างความเสียหายได้มากกว่าที่คาดคิดหลายเท่าตัว และความหรูหรานั้นไม่มีใครมองเห็น ไม่มีใครตีคุณค่า จะมีก็แต่ตัวเองเพียงผู้เดียว

สิ่งที่ช่วยวางอัลกอริทึมได้ดี

ความสามารถในการจำ ความสามารถในการแยกเป็นกรณีต่าง ๆ ให้ครบถ้วน และความสามารถในการคาดการณ์ล่วงหน้านับเป็นคุณสมบัติที่สำคัญสำหรับคนที่จะสร้างอัลกอริทึม *"ความจำ"* นั้นจำเป็นเพราะการวางแผนเป็นขั้นตอนนั้นเราต้องมองให้เห็นภาพว่าจะเกิดอะไรขึ้นเป็นขั้น ๆ ตัวเลขหรือข้อมูลต่าง ๆ จะถูกเปลี่ยนไปตามจังหวะ จึงต้องจดจำให้ได้ว่าเมื่อมาถึงแต่ละขั้นตอนข้อมูลจะเปลี่ยนรูปไปอย่างไร เพื่อมิให้เกิดความสับสน

"การแยกประเด็น" หรือแยกเรื่องออกเป็นกรณีต่าง ๆ นั้นใช้สำหรับแยกส่วนย่อยของโปรแกรม วิธีการเขียนโปรแกรมต้องครอบคลุมทุกกรณีที่จะเป็นไปได้ นอกจากนี้แต่ละกรณีที่จะเกิดขึ้นนั้นเราต้องมองเห็นได้ชัดเจนว่ามันแตกต่างหรือเหมือนกันในเรื่องใดบ้าง

"ความสามารถในการคาดการณ์ล่วงหน้า" จะมีประโยชน์ในการเลือกหนทาง การแก้ปัญหาแต่ละเรื่องทำได้

ร้อยแปดพันวิธี เราต้องเลือกวิธีที่ก่อให้เกิดปัญหาน้อยที่สุด

เขียนโปรแกรมให้ถูกวิธี

โปรแกรมกับอัลกอริทึมที่วางไว้จะต้องสอดคล้องและตรงกัน นี่คือหลักการที่สำคัญที่สุด ความหมายก็คือเวลาเขียนโปรแกรมนั้นให้เขียนเป็นส่วน ๆ ตามที่เราแยกเอาไว้เป็นข้อ ๆ ในอัลกอริทึม จงอย่าสอดส่ายสายตาควานหาส่วนอื่น ๆ ของโปรแกรมที่คล้ายกันแล้วใช้คำสั่ง GOTO เพื่ออาศัยข้อความที่เขียนไว้แล้ว เพราะวิธีเช่นนั้นจะทำให้แก้ไขยากในภายหลัง พึงสังเกตว่าการใช้คำสั่ง GOTO เพื่อให้สามารถใช้ส่วนของโปรแกรมซ้อนกันนี้แตกต่างจากการใช้รูทีนย่อย (Sub-routine) เพราะรูทีนย่อยนั้นเขียนไว้แยกต่างหาก ตัวที่เรียกมาใช้งานจะแยกกันอยู่ในแต่ละส่วนต่าง ๆ ของอัลกอริทึม ทำให้ทุกขั้นตอนในอัลกอริทึมไม่ปะปนกัน การใช้รูทีนย่อยจึงทำได้

ดังที่ได้กล่าวมาแล้วคนเรานั้นสามารถสร้างอัลกอริทึมที่ยาว 10 ถึง 50 คำสั่งให้ถูกต้องได้ไม่ยาก แต่มักจะสร้างผิดเมื่อมีหลายขั้นตอนเกินไป ดังนั้นหลังจากที่ได้พยายามเขียนบางส่วนของโปรแกรมลงไปแล้ว เราควรตรวจดูว่ามีส่วนไหนที่ยาวเกิน 50 บรรทัดบ้าง ถ้ามีอยู่หรือมีบางขั้นตอนที่น่าสงสัยว่าอาจเขียนผิดก็ต้องแก้ไข เราต้องกลับไปทบทวนอัลกอริทึมใหญ่ของเรา มันอาจต้องแยกเป็นขั้นตอนย่อยในขั้นตอนใหญ่หรือวางโครงร่างกันใหม่หมดเลยก็เป็นได้ เรื่องนี้ต้องทำการทบทวนอยู่เสมอ ๆ จะปล่อยให้โปรแกรมแต่ละขั้นตอนยาวเกิน 50 คำสั่งไม่ได้ นี่ก็เป็นหัวใจของการเขียนโปรแกรมอีกอย่างหนึ่ง

เรื่องง่าย ๆ แต่ทำไมสอนยาก

หลักการที่กล่าวมานี้ดูง่ายเกินไป นี่คือคำตอบ โปรแกรมเมอร์ส่วนใหญ่ไม่สนใจที่จะเขียนอัลกอริทึมลงบนกระดาษแล้วทบทวนมันให้ดีเสียก่อน พวกเขาใจร้อนและจะลงมือเขียนโปรแกรมไปเลย เรื่องนี้เราจะบอกว่าพวกเขาไม่มีการวางอัลกอริทึมก็ไม่เชิง เพราะความจริงแล้วเขามีแนวคิดอยู่ เป็นการวางขั้นตอนอย่างคร่าว ๆ ในหัวสมอง แล้วเขาก็พยายามถ่ายทอดออกมาเป็นโปรแกรม แต่เหตุที่ทำให้เขาเขียนโปรแกรมผิดบ่อย ๆ เกิดจากขั้นตอนในสมองของเขามีน้อยเกินไป เมื่อแตกออกมาเป็นคำสั่งในโปรแกรมอาจจะยาวเกิน 50 คำสั่งต่อ 1 ขั้นตอนแล้วมันก็เลยผิดพลาดบ่อย ๆ

สรุปท้ายเรื่อง

ปัญหาการเขียนโปรแกรมแล้วมีบั๊กต้องแก้ไขอยู่เป็นเวลานาน ๆ นั้นมาจากความใจร้อน การจะเขียนโปรแกรมให้ได้ดีนั้นต้องวางเป็นขั้นตอนใหญ่ ๆ เสียก่อน ซึ่งเราเรียกว่าอัลกอริทึม จากนั้นก็มาแตกขั้นตอนใหญ่ออกเป็นขั้นตอนย่อย ๆ เราจะต้องแยกไปจนถึงระดับที่สามารถเขียนเป็นโปรแกรมไม่เกิน 50 คำสั่งต่อ 1 ขั้นตอนย่อย หลังจากนั้นจึงจะลงมือเขียนโปรแกรม การเขียนโปรแกรมเป็นการถ่ายทอดจากขั้นตอนต่าง ๆ ในอัลกอริทึมทีละขั้นตอน ทุกอย่างต้องสอดคล้องและตรงกัน จะไม่มีการใช้ส่วนหนึ่งส่วนใดของโปรแกรมซ้ำซ้อนกัน ยกเว้นในกรณีที่เห็นได้เด่นชัดว่าแยกเป็นรูทีนย่อยแล้วจะประหยัดและเข้าใจง่ายขึ้น เรื่องนี้จริง ๆ แล้วง่ายมาก แต่ก็สอนกันยาก คนส่วนใหญ่ยังชอบยึดติดมั่นกับแนวคิดของตนเอง